

Add new user-song associations in batches, allowing a significant period of time between each batch.

- 5 Since the total that is in the denominator of all the p calculations will not change in between batches, that makes it possible, at the end of a batch load, to create a one-dimensional array to represent the p log p values, where the index is the numerator in the p calculation. Thus, each relevant p log p calculation only needs to be performed once, and is then reused.
- 10 Instead of actually re-allocating memory for the array at the end of each batch load, the array can be zeroed out. A 0 in an element indicates that p log p has not yet been calculated. So, when a value is needed for p log p, the appropriate element is checked, and if it is 0, it is calculated. If it is non-zero, then the value that is there is used.

Appendix C

```
15 #VERSION 12 08/27/00

#Copyright (c) 2000 by Virtual Development Corp. All Rights Reserved.

#Usage Notes#####
# MiminimumConvergenceIterations in the Config file must be at least 1. (See BUGS.)

# MinimumConvergenceIterations "beats" MaxTime. It will run for the minimum
#   number of configurations, then run until MaxTime.

20
25

# work_ = Work instance
# rel_ = Relatable instance
# clus_ = Cluster
# clst_ = ClusterSet
# clss_ = ClusterSetSignature

30
35

import whrandom
import math
import xmllib
import copy
import time
import ConfigParser
import urllib
```

```
import sys

5
# Utility stuff
G_generator = whrandom.whrandom() # For why global, see
http://starship.python.net/crew/donp/script/sample.py
#G_generator.seed(1,1,3)

10
def shuffle(sample_size):      # See
http://starship.python.net/crew/donp/script/sample.py
    '''Moses and Oakford algorithm.  See Knuth, vol 2, section 3.4.2.
    Returns a random permutation of the integers from 1 to
15    sample_size.
    '''
    assert type(sample_size) == type(0) and sample_size > 0
    global G_generator
    list = range(1, sample_size + 1)
G20
    for ix in xrange(sample_size - 1, 0, -1):
        rand_int = G_generator.randint(0, ix)
        if rand_int == ix:
            continue
        tmp = list[ix]
        list[ix] = list[rand_int]
        list[rand_int] = tmp
    return list

25
# from http://starship.python.net/pipermail/python-de/1997q1/000026.html
# "Converter module from strings to HTML entities"
# The code is modified slightly modified to use the encodings
# the python xml parser defaults to decoding, rather than using
# htmlentitiesdefs.

30
# 35 EntitiesByOrd={ ord('<') : 'lt',
#                   ord('>') : 'gt',
#                   ord('&') : 'amp',
#                   ord('"') : 'quot',
#                   ord("'") : 'apos' }

40
def toXML(s):
    pos=start=0
    result=""
    flush=0
45
    while pos<len(s):
        c=ord(s[pos])
```

```
if EntitiesByOrd.has_key(c):
    flush=1
    item="&"+EntitiesByOrd[c]+";"
if flush:
    5      result=result+s[start:pos]+item
    start=pos+1
    flush=0
    pos=pos+1
    result=result+s[start:pos]
10     return result

def computeEvenRankUnitRanks( lstTup_input ):
    # SHOULD BE IN DATA object

15     # Suppose 100 values are tied for second place, and 1
    # is alone in first. It should not be assumed that we
    # should put the lone value in the top percentile, because
    # it could easily be due to noise. So, we compromise by
    # saying there are 2 ranks, and we assign .25 to everyone in the low
20     # and .75 to the one in the high.

25     # We only use the first element in the tuple for ranking.

30     # Output list has the same data as the input, but in
    # rank order, and each tuple has two extra elements
    # at the end: the integer rank (ties are counted as
    # the same rank; best is highest) and the unit rank.

35     # FURTHER ADJUSTMENT DURING TIME OF LITTLE DATA!!!! If
    # there are two input sort field values, 1 and 2, the
    # original algorithm gives outputs .25 and .75. But that
    # still means that the low level is much closer to 0
    # than the high level is. That makes no sense.
    # So, we change the levels to .625 and .875.

40     lstTup_input.sort()
        assert lstTup_input[ 0 ][ 0 ] != None    # logic assumes first sort value is not None
        lstTup_intermediate = []
        int_rank = 0
        any_previousSortValue = None
        for tup_ in lstTup_input:
            if any_previousSortValue != tup_[ 0 ]:
                int_rank = int_rank + 1
45            any_previousSortValue = tup_[ 0 ]
            lstTup_intermediate.append( tup_ + ( int_rank, ) )

        lstTup_output = []


```

```
for tup_ in lstTup_intermediate:
    float_ = ( tup_[ -1 ] - 0.5 ) / float( int_rank )
    float_tuning = Config.float_tuningRankBottom + float_ * ( 1.0 -
Config.float_tuningRankBottom ) #see note above for little data
5     lstTup_output.append( tup_ + (float_tuning,) )

return lstTup_output

10

def computeAverageUnitRanks( lstTup_input ):
    # NOT USED IN CURRENT CODE 8/24/00
    # The first element in the tuple is the only one used
    # in the ranking.
15    # The output list contains tuples identical to the input
    # list but with an added element at the end, which is
    # the ranking, with dups assigned to the average ranks
    # of the dups.

20    def isLastInDupSet( int_index, lstTup_ ):
21        if len( lstTup_ ) == int_index + 1:
22            return 1
23        else:
24            if lstTup_[ int_index ][ 0 ] != lstTup_[ int_index + 1 ][ 0 ]:
25                return 1
            else:
                return 0

26        float_offset = 1.0 / ( 2.0 * len( lstTup_input ) )
27        lstTup_input.sort()
28        lstTup_output = []
29        int_startDupIndex = 0
30        int_limitIndex = len( lstTup_input )
31        lst_currentDupSet = []
32
33        for int_index in range( int_limitIndex ):
34            if isLastInDupSet( int_index, lstTup_input ):
35                lst_currentDupSet.append( lstTup_input[ int_index ] )

36                # Compute average unit rank
37                float_averageRank = ( int_index + int_startDupIndex ) / 2.0
38                float_averageUnitRank = float_offset + float_averageRank / int_limitIndex

39
40                # Add to output list
41                for tup_ in lst_currentDupSet:
42                    lstTup_output.append( tup_ + ( float_averageUnitRank, ) )

43
44                # Set the stage for next iteration
45                int_startDupIndex = int_index + 1
```

```
    lst_currentDupSet = []
else:
    lst_currentDupSet.append( lstTup_input[ int_index ] )

5      return lstTup_output

# Classes
10
class Config:
    # When an instance is created, the class attributes are populated;
    # at that point, the instance itself can be thrown away.

15      str_clusterFile = None
      str_useFile = None
      str_oldUseFile = None
      int_createClusterCount = None
      float_maxTime = None
20      int_minimumConvergenceIterations = None
      str_outClusterFile = None
      float_tuningRankBottom = None
      float_tuningZeroWeight = None

25      C_str_configFile = 'clusterconfig.txt'
      C_str_sectionName = 'Configuration'
      C_str_clusterFile = 'InClusterFile'
      C_str_useFile = 'UseFile'
      C_str_oldUseFile = 'OldUseFile'
30      C_str_createClusterCount = 'CreateClusterCount'
      C_str_maxTime = 'MaxTime'
      C_str_minimumConvergenceIterations = "MinimumConvergenceIterations"
      C_str_outClusterFile = 'OutClusterFile'
      C_str_tuningRankBottom = 'TuningRankBottom'
35      C_str_tuningZeroWeight = 'TuningZeroWeight'

    def __init__( self ):
        configParser = ConfigParser.ConfigParser()
        configParser.read( Config.C_str_configFile )
40        Config.str_clusterFile = configParser.get( Config.C_str_sectionName,
Config.C_str_clusterFile )
        Config.str_useFile = configParser.get( Config.C_str_sectionName,
Config.C_str_useFile )
        Config.str_oldUseFile = configParser.get( Config.C_str_sectionName,
Config.C_str_oldUseFile )
45        Config.int_createClusterCount = int( configParser.get( Config.C_str_sectionName,
Config.C_str_createClusterCount ) )
```

```
Config.float_maxTime = float( configParser.get( Config.C_str_sectionName,
Config.C_str_maxTime ))
Config.int_minimumConvergenceIterations = int( configParser.get(
Config.C_str_sectionName, Config.C_str_minimumConvergenceIterations ))
5   Config.float_tuningRankBottom = float( configParser.get( Config.C_str_sectionName,
Config.C_str_tuningRankBottom ))
   Config.float_tuningZeroWeight = float( configParser.get( Config.C_str_sectionName,
Config.C_str_tuningZeroWeight ))
   Config.str_outClusterFile = configParser.get( Config.C_str_sectionName,
10  Config.C_str_outClusterFile )

class Data:
    # This is a singleton. One instance is created, and that creates everything.
15
    # "Longnames" are of the format "Beatles - Hey Jude". The artist and the title
separate by
    # spacedashspace. Each Work object is uniquely identified by a Longname.

20 singleton = None

25
def __init__( self ):
    assert not self.__class__.singleton
    self.__class__.singleton = self
    self.dictStrDictStrNone_userLongname = {}
    self.dictStrDictStrFloat_longname2Longname1UnitRank = {}
    self.dictLongnameWork_ = {}
    self.dictStrDictStrInt_longname1Longname2Count = {}
    self.dictStrInt_longnameUniqueCount = {}
30    self.lstWork_ = []

35
    assert Config.str_useFile
    print 'about to read data'
    self.__readUserPlayStats( Config.str_useFile )
    print 'about to generate use counts'
    self.__generateUseCounts()
    print 'about to generate unit ranks'
    self.__generateUnitRanks()

40
def displayCheckingInfo( self ):
    dict_russians = self.dictStrDictStrFloat_longname2Longname1UnitRank[ 'Sting -
Russians' ]
    lst_russians = dict_russians.items()
    lst_russians.sort()
45

def getWorks( self ):
```

```
    return self.lstWork_

def getUnitRanks( self ):
    assert self.dictStrDictStrFloat_longname2Longname1UnitRank
5     return self.dictStrDictStrFloat_longname2Longname1UnitRank

def getAssociatedLongnames( self, str_longname ):
    assert self.dictStrDictStrFloat_longname2Longname1UnitRank.has_key( str_longname )
    return self.dictStrDictStrFloat_longname2Longname1UnitRank[ str_longname ].keys()
10

def __readUserPlayStats( self, str_fileName ):
    if str_fileName[ :7 ] == "http://":
        fil_ = urllib.urlopen(str_fileName)
15    else:
        fil_ = open(str_fileName, 'r')
        str_ = fil_.read()
        fil_.close()

20 class UseListContainerParser1( xmlllib.XMLParser ): # Embedded class, only used
here!
    # THIS LOGIC ASSUMES UNIQUENESS AT USER/SONG LEVEL IN THE INPUT XML FILE!!

25     def __init__( self, data_ ):
        self.str_currentUser = None
        self.data_ = data_
        xmlllib.XMLParser.__init__( self )

30     def start_entry( self, dict_ ):
        # str_work is the title of the work, which must be distinguished from Work
objects!
35
        if ( self.str_currentUser != 'mike3k@mail.com'
            and self.str_currentUser != 'jake@jspace.org'
            and self.str_currentUser != 'jake@braincase.net' ):
            if int( dict_[ 'count' ] ) > 1:
                str_artist = intern( dict_[ 'artist' ] )
                str_work = intern( dict_[ 'work' ] )
                str_longname = intern( '%s - %s' % ( str_artist, str_work ) )

40            dict_ = self.data_.dictStrInt_longnameUniqueCount
            if dict_.has_key( str_longname ):
                dict_[ str_longname ] = dict_[ str_longname ] + 1
            else:
                dict_[ str_longname ] = 1

45        def start_useList( self, dict_ ):
```

```
    self.str_currentUser = dict_[ 'user' ]\n\n    class UseListContainerParser2( xmllib.XMLParser ): # Embedded class, only used\n    here!\n5\n    def __init__( self, data_ ):\n        self.str_currentUser = None\n        self.data_ = data_\n        xmllib.XMLParser.__init__( self )\n\n10\n    def start_entry( self, dict_ ):\n        # str_work is the title of the work, which must be distinguished from Work\n        objects!\n        str_artist = intern( dict_[ 'artist' ] )\n15\n        str_work = intern( dict_[ 'work' ] )\n        str_longname = intern( '%s - %s' % ( str_artist, str_work ) )\n\n        if ( self.data_.dictStrInt_longnameUniqueCount.has_key( str_longname ) and\n            self.data_.dictStrInt_longnameUniqueCount[ str_longname ] > 1 ):\n\n20\n            if self.data_.dictStrDictStrNone_userLongname.has_key( self.str_currentUser ):\n                if self.data_.dictStrDictStrNone_userLongname[ self.str_currentUser\n].has_key( str_longname ):\n                    pass      # Already there!\n                else:\n                    self.data_.dictStrDictStrNone_userLongname[ self.str_currentUser ][\nstr_longname ] = None\n                else:\n                    self.data_.dictStrDictStrNone_userLongname[ self.str_currentUser ] = {\nstr_longname : None }\n\n25\n\n30\n            if not self.data_.dictLongnameWork_.has_key( str_longname ):\n                work_ = Work( str_longname, str_artist, str_work )\n                self.data_.lstWork_.append( work_ )\n                self.data_.dictLongnameWork_[ str_longname ] = work_\n\n35\n\n        def start_useList( self, dict_ ):\n            self.str_currentUser = dict_[ 'user' ]\n\n\n40\n        parser_1 = UseListContainerParser1( self )\n        parser_1.feed( str_ )\n        parser_1.close()\n        parser_2 = UseListContainerParser2( self )\n        parser_2.feed( str_ )\n45\n        parser_2.close()\n\n\n        def __generateUseCounts( self ):\n            dictStrDictStrInt_longname1Longname2Count = {}
```

```
1 lstStr_user =self.dictStrDictStrNone_userLongname.keys()
2     int_loopCount = 0
3     for str_user in lstStr_user:
4         int_loopCount = int_loopCount + 1
5
6         int_innerLoopCount = 0
7         sys.stdout.flush()
8         for str_longname1 in self.dictStrDictStrNone_userLongname[ str_user ].keys():
9             int_innerLoopCount = int_innerLoopCount + 1
10            # print 'deep in loop, ', int_innerLoopCount, ' of ',
11            len(self.dictStrDictStrNone_userLongname[ str_user ])
12            for str_longname2 in self.dictStrDictStrNone_userLongname[ str_user ].keys():
13                # if str_longname1 != str_longname2: songs played by only 1 user can still be
14                # clustered due
15                #                                         to the user's other choices...
16                not counting cases
17                #                                         where the two are equal would
18                # eliminate them, and
19                #                                         should cause logic that loops
20                # through all of the songs
21                #                                         looking for unitRanks to fail
22
23                if str_longname1 != str_longname2:
24                    if dictStrDictStrInt_longname1Longname2Count.has_key( str_longname1 ):
25                        if dictStrDictStrInt_longname1Longname2Count[ str_longname1 ].has_key(
26                            str_longname2 ):
27                            dictStrDictStrInt_longname1Longname2Count[ str_longname1 ][ str_longname2
28                            ] = \
29                                dictStrDictStrInt_longname1Longname2Count[ str_longname1 ][
30                                str_longname2 ] + 1
31                                else:
32                                    dictStrDictStrInt_longname1Longname2Count[ str_longname1 ][ str_longname2
33                                    ] = 1
34                                else:
35                                    dictStrDictStrInt_longname1Longname2Count[ str_longname1 ] = {
36                                        str_longname2 : 1 }
37
38        self.dictStrDictStrInt_longname1Longname2Count =
39        dictStrDictStrInt_longname1Longname2Count
40
41
42    def __generateUnitRanks( self ):
43        # "Unit ranks" are ranks scaled down to the unit interval. For instance, the lowest
44        # rank out of 57 elements is 0, and the highest is 56/57 = .98245614035. But, we
45        # also perform averaging, so ranks that extreme should be unusual.
46
47        # Consider longname1 to be a work "associated" with longname2. Longname2 is the
48        work
49        # for which we are generating a profile; this profile involves the
50        associated
51        # Longname1 works.
```

```
# That is, a profile for a longname2 would contain all
# the longname1's that are associated with it. For each associated work, considered
across all
# main works, there is one rank for each main work,
5      # that's where the uniform distribution comes from. The alternative would be: for
each main work have
# one rank for each associated work; then some associated works would NECESSARILY
have very low rank.
# In contrast, using the approach presented, all associated works CAN have high
10     rank -- but under
# the null hypothesis the distribution would be uniform.

self.dictStrDictStrFloat_longname2Longname1UnitRank      = {}
15    for str_longname1 in self.dictStrDictStrInt_longname1Longname2Count.keys():
        lstTupIntStr_ = []
        dictStrInt_longname2Count = self.dictStrDictStrInt_longname1Longname2Count[
str_longname1]
        for str_longname2 in dictStrInt_longname2Count.keys():
            lstTupIntStr_.append(( dictStrInt_longname2Count[ str_longname2 ], str_longname2
))
20
25    if str_longname1 == 'Elton John - Levon':
        lstTupIntStr_.sort()
        lstTupIntStrIntFloat_ = computeEvenRankUnitRanks( lstTupIntStr_ )
        for int_ in range( len( lstTupIntStrIntFloat_ ) ):
            tupIntStrIntFloat_ = lstTupIntStrIntFloat_[ int_ ]
            float_ = tupIntStrIntFloat_[ -1 ]
            str_longname2 = lstTupIntStrIntFloat_[ int_ ][ 1 ]
            if self.dictStrDictStrFloat_longname2Longname1UnitRank.has_key( str_longname2 ):
                self.dictStrDictStrFloat_longname2Longname1UnitRank[ str_longname2 ][
str_longname1 ] = float_
            else:
                self.dictStrDictStrFloat_longname2Longname1UnitRank[ str_longname2 ] = {
str_longname1 : float_
}
30
35    # fil_.close()

# computeAverageUnitRanks

40    class Relatable:
        def getName( self ):
            assert 0

        def getAssociatedRelatedness( self, str_otherName ):
45            assert 0

        def getAssociatedLongnames( self ):
            assert 0
```

```
def getOverallRelatedness( self, rel_ ):

    float_zeroWeight = Config.float_tuningZeroWeight
5
    float_sum = 0.0
    float_divisor = 0.0
    for str_name in self.getAssociatedLongnames():
        float_other = rel_.getAssociatedRelatedness( str_name )
10
        if float_other == None:    # Defensive programming
            float_other = 0.0
        if float_other == 0:
            float_weight = float_zeroWeight
        else:
            float_weight = 1.0
15
        float_divisor = float_divisor + float_weight
        float_self = float( self.getAssociatedRelatedness( str_name )) # Cast is
        defensive programming
        float_product = float_self * float_other * float_weight
20
        float_sum = float_sum + float_product
        if float_divisor:
            float_overallRelatedness = float_sum / float_divisor
        else:
            float_overallRelatedness = 0.0
25
    return float_overallRelatedness

30
class Work( Relatable ):
    # The xml attribute 'work' is the title of the work, which must be distinguished from
    Work objects,
    # which contain artist info as well as title info!

35
    def __init__( self, str_longname, str_artist, str_work ):
        # The "Longname" of the work, for purposes of this program, is the artist + the
        work title.
        Data.singleton.getAssociatedLongnames( str_longname )
        self.str_longname = str_longname
        self.str_artist = str_artist
        self.str_work = str_work

40
    def getName( self ):
        return self.str_longname

        def getArtist( self ):
            return self.str_artist
45
    def getAssociatedRelatedness( self, str_longname ):
        dictStrDictStrFloat_longname1UnitRank = Data.singleton.getUnitRanks()
```

```
dict_ = dictStrDictStrFloat_longname2Longname1UnitRank #Using intermediate name
just for clarity
    assert dict_.has_key( self.str_longname )
    if dict_[ self.str_longname ].has_key( str_longname ):
5        float_unitRank = dict_[ self.str_longname ][ str_longname ]
    else:
        float_unitRank = 0.0
    return float_unitRank

10   def getAssociatedLongnames( self ):
    return Data.singleton.getAssociatedLongnames( str_longname )

class Cluster( Relatable ):
    # To understand this class, it's important to understand the difference between a
15    # cluster's membership list and its profile. Both of them involve a group of
    # objects subclassed from Relatable. But the membership list (self.lstRel_member)
    # determines the objects that are currently members of a cluster; whereas, the
    # profile (self.dictStrFloat_longnameRelatedness) is a description of the current
    # "center" of the cluster for purposes of measuring the distance between the
    # cluster and an object that is a candidate for membership in the cluster.

20
    # Normally, all candidate objects are assigned to a cluster before the profile
    # is computed; these assignments are based on the old profiles. For instance,
    # when clusters are being generated for the first time, the old profiles are
    # random. When clusters are being regenerated based on old clusters read from
    # an xml disk file, the profiles from the disk file clusters are used as the
    # old profiles.

25
    str_nextAutomaticName = '1'

30   def __init__( self, str_name=None ):
        self.lstRel_member = []
        self.dictStrFloat_longnameRelatedness = {}
        if str_name:
            self.str_name = str_name
        else:
            35            int_ = int( self.__class__.str_nextAutomaticName )
            self.str_name = self.__class__.str_nextAutomaticName
            self.__class__.str_nextAutomaticName = str( int_ + 1 )

40
    def getName( self ):
        return self.str_name

45   def getMembers( self ):
        return self.lstRel_member

    def getAssociatedRelatedness( self, str_longname ):
        # 1 or 0
```

```
if self.dictStrFloat_longnameRelatedness.has_key( str_longname ):
    return self.dictStrFloat_longnameRelatedness[ str_longname ]
else:
    return 0.0
5

def getCountUniqueArtist( self ):
    if not self.lstRel_member:
        return 0
    assert self.lstRel_member[ 0 ].__class__ == Work
10
    dict_ = {}
    for work_ in self.lstRel_member:
        dict_[ work_.getArtist() ] = None
    return len( dict_ )

15
def getAssociatedLongnames( self ):
    return self.dictStrFloat_longnameRelatedness.keys()

def addToCluster( self, rel_ ):
    self.lstRel_member.append( rel_ )

20
def addToProfile( self, strLongname ):
    # Used for initializing empty profile for later clustering.
    self.dictStrFloat_longnameRelatedness[ strLongname ] = None

25
def computeClusterProfile( self, bool_binary ):
    # Normally, relatedness of each member to the cluster is binary --
    # 1 if it's in the dict, 0 otherwise. However, in the final
    # cluster convergence, it makes sense to do a 2-stage profile computation;
    # first we compute the binary values (represented by membership in
    # the dict vs. non-membership), then, using those values, we recompute
    # the profile, generating floating point values. This allows
    # us, in the final convergence, to generate clusters in such
    # a way that the most remote profile elements don't hold as great a sway
    # over what potential members are attracted to the cluster.

30
    # WHILE REVIEWING THIS CODE FOR WORK ON CLUSTERS13, I NOTICED THAT THIS
    # APPARENTLY SHOULD BE STRUCTURED AS: IF BOOL_BINARY...ELSE. THIS WOULD
    # AVOID SETTING dictStrFloat_longnameRelatedness TWICE, AS APPARENTLY
40
    # HAPPENS WITH THE CURRENT CODE. NOT CHANGING NOW BECAUSE AM WORKING
    # ON NEW VERSION AND DO NOT EXPECT TO TEST CHANGES.

    for rel_ in self.lstRel_member:
        if rel_.__class__ == Work:
45
            self.dictStrFloat_longnameRelatedness[ rel_.getName() ] = 1.0
        elif rel_.__class__ == Cluster:
            lstStr_otherName = rel_.getAssociatedLongnames()
            for str_otherName in lstStr_otherName:
```

```
        self.dictStrFloat_longnameRelatedness[ str_otherName ] = 1.0
    else:
        assert 0 # Attempt to cluster an illegal class

5     if not bool_binary:
        for rel_ in self.lstRel_member:
            if rel_.__class__ == Work:
                self.dictStrFloat_longnameRelatedness[ rel_.getName() ] =
self.getOverallRelatedness( rel_ )
10    elif rel_.__class__ == Cluster:
        lstStr_otherName = rel_.getAssociatedLongnames()
        for str_otherName in lstStr_otherName:
            self.dictStrFloat_longnameRelatedness[ str_otherName ] =
self.getOverallRelatedness( rel_ )
15    else:
        assert 0 # Attempt to cluster an illegal class

20    def makeEmpty( self ):
        # Notice that it leaves the profile (self.dictStrFloat_longnameRelatedness) intact
for purposes
        # of getAssociatedRelatedness() and getAssociatedLongnames().

        self.lstRel_member = []
25
25    def merge( self ):
        # Turns a cluster of clusters (each of which must contain works)
        # into a cluster of works

30    lstWork_ = []

        for clus_ in self.lstRel_member:
            assert clus_.__class__ == Cluster
            for work_ in clus_.getMembers():
35            assert work_.__class__ == Work
            lstWork_.append( work_ )
        self.lstRel_member = lstWork_

35    class ClusterSet:
40        def __init__( self, str_fileName=None, 1stClus_persistent=None,
int_randomClusterCount=None ):
            # The constructor just loads or creates the clusters, it doesn't
            # do any processing.
            # When constructing from a file, the clusters
45            # have profiles for measuring relatedness, but have no members.
            # When constructing from a list of clusters, they keep their members.
            # Randomly generated clusters are given members.
            self.1stClus_ = []
```

```
if str_fileName:
    __readUserPlayStats( str_fileName )

5    elif int_randomClusterCount:
        lstWork_ = Data.singleton.getWorks()
        int_countWorks = len( lstWork_ )
        lstInt_shuffled = shuffle( int_countWorks )
        if int_countWorks < int_randomClusterCount: # Obviously only applicable in small
10   tests.
        int_randomClusterCount = int_countWorks
        int_numberOfRandomWorksPerCluster = int_countWorks / int_randomClusterCount
        clus_current = None
        for int_ in xrange( int_countWorks ):
15            if int_ % int_numberOfRandomWorksPerCluster == 0:
                if clus_current:                      #Skip first iteration
                    clus_current.computeClusterProfile( bool_binary=1 )
                clus_current = Cluster()
                self.addToClusterSet( clus_current )
                clus_current.addToCluster( lstWork_[ lstInt_shuffled[ int_ ] - 1 ] )
                clus_current.computeClusterProfile( bool_binary=1 ) # May end up doing this
20   twice for a cluster
                else:
                    assert lstClus_persistent
                    self.lstClus_ = lstClus_
25

def consolidateArtists( self ):
    # Move all works for a given artist to the cluster with the greatest
    #   concentration of works for that artist.

30    # This may not be necessary in implementations where can do all clustering at
        artist level.

        dictStrDictClusInt_artistClusterCount = {}
35    dict_ = dictStrDictClusInt_artistClusterCount # short handle

        for clus_ in self.lstClus_:
            lstWork_ = clus_.getMembers()
            for work_ in lstWork_:
40            str_artist = work_.getArtist()
            if dict_.has_key( str_artist ):
                if dict_[ str_artist ].has_key( clus_ ):
                    dict_[str_artist ][ clus_ ] = dict_[ str_artist ][ clus_ ] + 1
                else:
45            dict_[ str_artist ][ clus_ ] = 1
            else:
                dict_[ str_artist ] = { clus_ : 1 }
```

```
dictStrClus_artistBestCluster = {}

for str_artist in dict_.keys():
    clus_bestCluster = None
    5      int_bestCount = 0
    for tupClusInt_ in dict_[ str_artist ].items():
        if tupClusInt_[ 1 ] > int_bestCount:
            int_bestCount = tupClusInt_[ 1 ]
            clus_bestCluster = tupClusInt_[ 0 ]
    10     dictStrClus_artistBestCluster[ str_artist ] = clus_bestCluster

for clus_ in self.lstClus_:
    clus_.makeEmpty()

15     dictStrLstWork_artistWork = {}
    for work_ in Data.singleton.getWorks():
        str_artist = work_.getArtist()
        if dictStrLstWork_artistWork.has_key( str_artist ):
            dictStrLstWork_artistWork[ str_artist ].append( work_ )
        else:
            dictStrLstWork_artistWork[ str_artist ] = [ work_ ]
    20

for tupStrClus_ in dictStrClus_artistBestCluster.items():
    str_artist = tupStrClus_[ 0 ]
    clus_ = tupStrClus_[ 1 ]
    25      for work_ in dictStrLstWork_artistWork[ str_artist ]:
        clus_.addToCluster( work_ )

    for clus_ in self.lstClus_:
        clus_.computeClusterProfile()
    30

def getAverageSquaredUniqueArtist( self ):
    int_sum = 0
    for clus_ in self.lstClus_:
        35      int_count = clus_.getCountUniqueArtist()
        int_sum = int_sum + int_count**2.0

    return float( int_sum ) / len( self.lstClus_ )

40     def getAverageCountUniqueArtist( self ):
        int_sum = 0
        for clus_ in self.lstClus_:
            int_count = clus_.getCountUniqueArtist()
            int_sum = int_sum + int_count
    45

        return float( int_sum ) / len( self.lstClus_ )
```

```
def getMaxCountUniqueArtist( self ):
    int_max = 0
    for clus_ in self.lstClus_:
        5      int_count = clus_.getCountUniqueArtist()
        if int_count > int_max:
            int_max = int_count
    return int_max

10     def getMinCountUniqueArtist( self ):
        int_min = len( Data.singleton.getWorks() )
        for clus_ in self.lstClus_:
            int_count = clus_.getCountUniqueArtist()
            if int_count < int_min:
                15      int_min = int_count
    return int_min

20     def getMaxClusterSize( self ):
        int_maxSize = 0
        for clus_ in self.lstClus_:
            int_size = len( clus_.getMembers() )
            if int_size > int_maxSize:
                25      int_maxSize = int_size
        return int_maxSize

30     def getSignature( self ):
        # Returns a dictionary which is a signature of the cluster
        # Convenient since dicts can be tested for equality, don't need identity
        dictStrDictStrNone_longnameLongname = {}
        for clus_ in self.lstClus_:
            str_clusterName = clus_.getName()
            35      dictStrDictStrNone_longnameLongname[ str_clusterName ] = {}
            for str_associatedLongname in clus_.getAssociatedLongnames():
                dictStrDictStrNone_longnameLongname[ str_clusterName ][ str_associatedLongname ] =
= None
            return dictStrDictStrNone_longnameLongname
40

def performClustering( self, lstRel_item, bool_recluster=0, bool_binary=1 ):
    # bool_recluster means recluster items that are already clustered.

45     # For defensive programming, we copy the list object (nothing in the list is
copied)
    # so that, when we add to the list below, it doesn't have side effects
    # for calling methods which expect the list to be unmodified
```

```
lstRel_itemToCluster = copy.copy( lstRel_item )

5      if bool_recluster:
        for clus_ in self.lstClus_:
            for rel_ in clus_.getMembers():
                lstRel_itemToCluster.append( rel_ )

10     for clus_ in self.lstClus_:
            clus_.makeEmpty()                      # Leaves profile intact
15     for rel_ in lstRel_itemToCluster:
            float_bestRelatedness = 0.0           # default to no correlation
            clus_best = None
            for clus_ in self.lstClus_:
                float_currentRelatedness = clus_.getOverallRelatedness( rel_ )
                if float_currentRelatedness > float_bestRelatedness:
                    float_bestRelatedness = float_currentRelatedness
                    clus_best = clus_
            if float_bestRelatedness:      # IF 0 DOES NOT GO INTO A CLUSTER!!
                clus_best.addToCluster( rel_ )

20     clus_.computeClusterProfile( bool_binary )           # Prepare the cluster
center for use in further correlation

25     def convergeClusters( self, float_latestTime, int_minimumIterations, bool_binary=1 ):
30         # float_latestTime is latest time to start an iteration

35         float_currentTime = time.time()
            dict_oldSignature = None
            int_iterations = 0
            bool_done = 0
            while not bool_done:
                if int_iterations < int_minimumIterations or float_currentTime <=
float_latestTime:
                    print 'iterating:', int_iterations
                    self.performClustering( [], bool_recluster=1, bool_binary=bool_binary )
                    dict_newSignature = self.getSignature()
                    if dict_newSignature == dict_oldSignature:
                        print 'finishing convergence due to unchanged signatures'
40                    bool_done = 1
                    else:
                        dict_oldSignature = dict_newSignature
                        float_currentTime = time.time()
                        int_iterations = int_iterations + 1
45                    else:
                        print 'finishing due to timeout'
                        bool_done = 1
```

```
def merge( self ):

    for clus_ in self.lstClus_:
5        clus_.merge()

def getClusters( self ):
    return self.lstClus_

10   def addToClusterSet( self, clus_ ):
        self.lstClus_.append( clus_ )

def __readUserPlayStats( self, str_fileName ):
    # We do not put members into the clusters, we only populate the profiles.
15   self.lstClus_ = []
    fil_ = open(str_fileName, 'r')
    str_ = fil_.read()
    fil_.close()

20   class ClusterParser( xmllib.XMLParser ): # Embedded class, only used here!

        def __init__( self, clst_ ):
            self.clst_ = clst_
            self.clus_current = None
            xmllib.XMLParser.__init__( self )

25        def start_member( self, dict_ ):
            str_artist = intern( dict_[ 'artist' ] )
            str_title = intern( dict_[ 'work' ] )
            tupStrStr_artistTitle = ( str_artist, str_title )
            str_longname = intern( '%s - %s' % tupStrStr_artistTitle )
            self.clus_current.addProfile( str_longname )

30        def start_cluster( self, dict_ ):
            self.clus_current = Cluster( dict_[ 'name' ] )
            clst_.lstClus_.append( clus_current )

35        parser_ = ClusterParser( self )
            parser_.feed( str_ )
            parser_.close()

40        def writeToDisk( self, str_fileName ):
            fil_ = open( str_fileName, 'w' )
            fil_.write( '<?xml version="1.0" encoding="ISO-8859-1"?>\n' )
            fil_.write( """<ClusterContainer xmlns:xsi="http://www.w3.org/1999/XMLSchema-
45           instance"
                           xsi:noNamespaceSchemaLocation='ViewListContainer.xsd'>\n""")
            fil_.write( '      <clusters medium="music">\n' )
```

```
for clus_ in self.lstClus_:
    fil_.write( '          <cluster name="%s">\n' % clus_.getName())
    lstTup_ = []
    for work_ in clus_.getMembers():
        5      float_relatedness = clus_.getOverallRelatedness( work_ )
        tup_ = ( float_relatedness, toXML( work_.str_artist ), toXML( work_.str_work ) )
        lstTup_.append( tup_ )
    lstTup_.sort()
    lstTup_.reverse()
10   for tup_ in lstTup_:
        fil_.write( '              <member artist="%s" work="%s" relatedness="%s" />\n' %
        ( tup_[ 1 ], tup_[ 2 ], tup_[ 0 ] ) )
        fil_.write( '          </cluster>\n' )
        fil_.write( '      </clusters>\n' )
15   fil_.write( '</ClusterContainer>\n' )
    fil_.close()
```

```
20 #####
#####
# SCRIPT LOGIC
#
25 try:
#
30     Config()    # Get configuration data
#
35     Data()      # Create data singleton
#
40     if Config.int_createClusterCount:
        # See http://www.math.tau.ac.il/~nin/learn98/idomil/
        int_numberOfClusters = int( Config.int_createClusterCount * math.log(
        Config.int_createClusterCount ) )
        float_maxTime = time.time() + Config.float_maxTime
        float_mostFabulous = float( len( Data.singleton.getWorks() ) * len(
        Data.singleton.getWorks() ) )
        while time.time() < float_maxTime:
            45     float_maxTime1 = (float_maxTime - time.time()) *.33 + time.time()
            float_maxTime2 = (float_maxTime - time.time()) *.66 + time.time()
            float_maxTime1 = (float_maxTime - time.time()) *.50 + time.time()
            float_maxTime2 = (float_maxTime - time.time()) *1.0 + time.time()
            print 'In outer loop #####'
            print 'about to make cluster set'
            clst_1 = ClusterSet( int_randomClusterCount=int_numberOfClusters )
```

```
print 'about to perform first clustering'
clst_1.performClustering( [], 1 )
print 'about to perform first convergence'
clst_1.convergeClusters( float_maxTime1, Config.int_minimumConvergenceIterations
5  )
lstClus_1 = clst_1.getClusters()
clst_2 = ClusterSet( int_randomClusterCount=Config.int_createClusterCount ) # A
set of clusters of clusters
print 'about to perform second clustering'
10 clst_2.performClustering( lstClus_1, 0 ) # Make clusters of clusters
print 'about to merge'
clst_2.merge() # Change from clusters of clusters to clusters of works
print 'about to perform second convergence'
clst_2.convergeClusters( float_maxTime2, Config.int_minimumConvergenceIterations
15  )
clst_2.performClustering( [], 1, bool_binary=0 )
print 'about to perform third convergence'
clst_2.convergeClusters( float_maxTime1, Config.int_minimumConvergenceIterations,
bool_binary=0 )
20 float_fabulousness = clst_2.getAverageSquaredUniqueArtist()
print 'max unique:', clst_2.getMaxCountUniqueArtist(), ' min unique:',
clst_2.getMinCountUniqueArtist()
print ' avg unique:', clst_2.getAverageCountUniqueArtist(), ' fabulousness:',
float_fabulousness
25 if float_fabulousness < float_mostFabulous:
    fil_ = open('tuninginfo.txt', 'w')
    fil_.write('float_tuningRankBottom: ' + str( Config.float_tuningRankBottom ) +
'\n')
    fil_.write('float_tuningZeroWeight: ' + str( Config.float_tuningZeroWeight ) +
'\n')
30     fil_.write('float_fabulousness: ' + str( float_fabulousness ) + '\n')
    fil_.write('clst_2.getMaxCountUniqueArtist(): ' + str(
clst_2.getMaxCountUniqueArtist() ) + '\n')
    fil_.write('clst_2.getMinCountUniqueArtist(): ' + str(
clst_2.getMinCountUniqueArtist() ) + '\n')
35     fil_.write('clst_2.getAverageCountUniqueArtist(): ' + str(
clst_2.getAverageCountUniqueArtist() ) + '\n')

        fil_.close()
40     print '###FOUND NEW BEST###'
        print 'writing intermediate'
        float_mostFabulous = float_fabulousness
        clst_2.writeToDisk( 'intermediate.xml' )

45     clst_best = clst_2

elif Config.str_clusterFile:
    clst_cluster = ClusterSet( str_fileName=Config.str_clusterFile )
```

```
    clst_cluster.performClustering( Data.singleton.getWorks(), 0 )
    clst_actual.convergeClusters( Config.float_maxTime + time.time(),
Config.int_minimumConvergenceIterations )
    else:
5      assert 0, 'Invalid config file option'
      clst_best.writeToDisk( Config.str_outClusterFile )
      print 'done!'
    except Exception, str_:
      print 'ERROR'
10   print str_
      print '\n\nPress any key to abort:'
      sys.stdin.read(1)
```

15

Bibliography

Klir, George and Folger, Tina. *Fuzzy Sets, Uncertainty, and Information*. Englewood Cliffs, NJ: Prentice Hall, 1988.

20 Manly, Bryan F.J. *Multivariate Statistical Methods, A Primer, Second Edition*. London, England: Chapman & Hall, 1994.

Hedges, Larry V. and Olkin, Ingram. *Statistical Methods for Meta-Analysis*. San Diego, CA: Academic Press, 1985.

25 Snijders, Tom A. B., Maarten Dormaar, Wijbrandt H. van Schuur, Chantal Dijkman-Caes, and Ger Driessen [1990]. "Distribution of Some Similarity Coefficients for Dyadic Banary Data in the Case of Associated Attributes." *Journal of Classification*, 7, pp. 5-31.

<http://www.google.com>
<http://www.interbase.com/>
<http://www.napster.com>

30